

Building a Game Engine: A Tale of Modern Model-Driven Engineering

Victor Guana
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
guana@ualberta.ca

Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
stroulia@ualberta.ca

Vina Nguyen
Department of Computing Science
University of Alberta
Edmonton, AB. Canada
vina1@ualberta.ca

Abstract—Game engines enable developers to reuse assets from previously developed games, thus easing the software-engineering challenges around the video-game development experience and making the implementation of games less expensive, less technologically brittle, and more efficient. However, the construction of game engines is challenging in itself; it involves the specification of well defined architectures and typical gameplay behaviors, flexible enough to enable game designers to implement their vision, while, at the same time, simplifying the implementation through asset and code reuse. In this paper we present a set of lessons learned through the design and construction PhyDSL-2, a game engine for 2D physics-based games. Our experience involves the active use of modern model-driven engineering technologies, to overcome the complexity of the engine design and to systematize its maintenance and evolution.

I. INTRODUCTION

Building video games is an effort-intensive and error-prone software-development task [1] [2]. In recent years, landmark papers have described the need of tackling the complexity of implementing engaging video-game designs [3] [4] and, to that end, a variety of game engines have been developed to provide developers with environments in the context of which to reuse gameplay production assets [5]. Such environments favour the reuse of off-the-shelf components, thus avoiding the need of building software assets from the ground up for individual titles [5]. Furthermore, game engines ease many of the software-engineering challenges around the video-game construction experience. They make the implementation of complex gameplay designs less expensive, less technologically brittle, and significantly more efficient. In this paper, we argue that many of these advantages stem not only from the active reusability policies that game engines promote, but from an effective use of game-authoring environments that facilitate the translation of gameplay mental models into computational artifacts that can be compiled and executed.

In [6], Johnson-Laird presented mental models as the main concept behind human reasoning. Johnson-Laird argued that mental models are key to capture human perception, imagination, and structural understanding of reality. In the context of video-game development, game developers implement gameplay designs using mostly general-purpose programming languages. In this process, game developers have to translate their gameplay mental models into the operational semantics

of a programming language. This task is cognitively challenging and often frustrating for developers, since general programming languages are not designed to capture the gameplay mechanics in the developers' vision of the game. The semantic gap between the developers' mental model of a game and the implementation artifacts that make this model executable, poses significant challenges to developers when reflecting about a gameplay design. This includes, but is not limited to, the developer's ability to reason about how faithful is the implementation of a game in relation to the developer's original vision of the title. We believe it is the role of the game-engine architects to provide adequate development interfaces, and to enable game designers to specify gameplay designs using semantics analogous to their gameplay mental model.

The software engineering principles behind the implementation of game engines have not been properly documented. This is mostly due to the proprietary and non-disclosure nature of this type of software systems [3]. However, Bishop [5] and Gregory [7] describe generic features that any game engine should support, namely, scene and rendering management, collision and rigid body mechanics, and human interface management. In this paper we extend the aforementioned feature set, and propose a generalizable architecture for 2D physics-based games.

The contribution of this paper is twofold. We first report how we designed and implemented a domain-specific language that enables developers to naturally define mental models of 2D physics-based games using high-level semantics, well aligned with the gameplay mental models of the game designers. Moreover, we describe how using model-driven engineering technologies, we implemented a translation mechanism that takes high-level gameplay specifications, and derives executable code for Android devices. We have packaged the proposed domain-specific language, and the translation mechanism, in a game engine called PhyDSL-2. Throughout the paper we share a list of propositions that reflect on our experiences building PhyDSL-2.

The rest of this paper is organized as follows, Section 2 presents the process of designing PhyDSL-2 domain-specific language; Section 3 discusses the architecture of the games that can be produced using PhyDSL-2; Section 4 describes how we used modern model-driven engineering tools, to gener-

ate concrete game implementations from high-level gameplay models; Finally, Section 5 summarizes our work.

A. The PhyDSL-2 Language

Proposition 1: Domain-specific languages enable developers to reflect about their gameplay mental models. They guide the specification, study, and optimization of game designs. DSLs are effective tools to capture the specification of a software system using semantics that belong to the system's domain [8] [9]. They are usually built on top of syntax-directed editors, and help developers to specify software systems using language constructs, analogous to the developers' mental model of a problem. Furthermore, DSLs can be tailored to match the mental models of a well defined development population. Mernik et al. [10] have studied how the proper scope definition of a DSL boosts the ability of developers to analyse, design, and optimize software specifications. In the context of PhyDSL-2, we focused on providing a game-development language for non-programming experts, yet appealing to professional software developers. Furthermore, we decided to limit the scope of the game engine to 2D-physics based games, which comprises a large portion of the casual and serious game markets¹.

In [11] we presented a detailed description of the PhyDSL language, and how it compares to the other game-authoring environments for non-programming experts. PhyDSL-2 is the evolution of PhyDSL, it provides new camera behaviors for following different types of game actors on the screen, and new on-screen control constructs for locating actionable elements and manipulate gameplay objects. PhyDSL-2 editor is built as a development plugin for Eclipse², a widely adopted integrated development environment (IDE). The language comes with a declarative textual syntax focused in the expression of gameplay designs, friendly to both programming and non-programming experts.

In order to define the scope of the PhyDSL-2 language, we considered five exploratory questions that game developers have to answer while creating a gameplay design in single-scene and scrollable physics-based games: (i) *who is the player?*, (ii) *what are the player's available actions?*, (iii) *where does the player live?*, (iv) *what are the player's goals?*, and (v) *what are the player's challenges?*. The ultimate goal of defining the scope of the language is to provide a set of constructs so developers can answer these gameplay-design questions, using semantics close to the level of abstraction of their gameplay vision. Furthermore, we strongly believe that our five exploratory questions can be generalized to help in the scope definition of any game development interface.

Our five gameplay-definition questions are aligned with the *meaningful play principles* summarized by Salen and Zimmerman in [12]. The principles argue that *"meaningful play in a game emerges from the relationship between player action and system outcome; it is the process by which a player*



Fig. 1. Alien Miner Gameplay

takes action within the designed system of a game and the system responds to the action. The meaning of an action in a game resides in the relationship between action and outcome." Moreover, Salen and Zimmerman argue that *"meaningful play occurs when the relationships between actions and outcomes in a game are both discernable and integrated into the larger context of the game"*. Let us now discuss how our exploratory questions help in the definition of the language scope.

Q1: "Who is the player?" explores the mechanisms available for the game designer to define the representation of the player in the game. For certain single-scene games, such as Bejeweled³ for example, the representation of the player is external, that is, the player interacts with the game using input mechanisms such as touching, or clicking on a gameplay element, and the input events directly modify the state of the game. In other games, such as Angry Birds⁴ for example, the representation of the player is internal and the player changes the state of the game indirectly by manipulating an avatar.

Q2: "What are the player's actions?" investigates how the game designer can specify the actions available to the player, for interacting with the game world to work through the gameplay challenges, and achieve their goals. A few examples of the player's available actions are *"hitting a button to open a door," "hitting a button to move the player in different directions,"* and *"touching a game element to move it"*.

Q3: "Where does the player live?" considers how to enable developers to build the game environment, in the context of which the players will be working towards achieving the gameplay goals.

Q4: "What are the player's goals?" scopes out how developers will be able to define gameplay milestones for the title under construction. The definition of goals for the game players constitutes the fundamental driving components of meaningful play [12]. Furthermore, this question also includes how developers can specify reward mechanisms for the player, whether it is through points, badges, items, power-ups, or simply the progress to another game level. Some goal

¹<http://www.newzoo.com/trend-reports/mobile-games-trend-report/>

²<http://eclipse.org/>

³<http://bejeweled.wikia.com/>

⁴<http://angrybirds.wikia.com/>

```

1  define AlienMiner game:
2      define mobile actors:
3          actor player (density: stone, elasticity : baseball, friction : none, render: alien, main:yes)
4          actor fireball (density: beachBall, elasticity : beachBall, friction : gel, render: fireball )
5      define static actors:
6          actor brick (density: stone, elasticity : stone, friction : none, render: granite )
7          actor portal (density: none, elasticity : none, friction : none, render: portal )
8          actor emerald (density: none, elasticity : none, friction : none, render: emerald)
9      define layout:
10         grid is (50, 20)
11             locate player (0, 10)
12             locate brick (10,10) ... more platforms
13             locate emerald(11, 11) ... more emeralds
14      define environment:
15         gravity is like moon
16         background is like cave
17         camera follows player continuously: yes
18      define activities :
19         activity cannon link to fireball {
20             appear (frequency:high, linear speed:slow, position : (15, 12))
21         }
22      define scoring rules:
23         rule reachPortalGoal {
24             collision of player with portal (points: +100, game ends: yes)
25         }
26         rule emeraldCollectionGoal {
27             collision of player with emerald (points: +10, game ends: no)
28         }
29         rule fireballHitPenalty {
30             collision of player with fireball (points: -20, game ends: yes)
31         }
32      define controls:
33         control buttonA (position : (0,10) render: aLabel){
34             player action UP
35         } ... more button definitions

```

Listing 1. PhyDSL-2 Alien Miner Definition

examples include, *"the player has to complete a course of obstacles and finish alive to obtain a badge,"* or *"the player has to collect elements and each element increases his/her score."*

Q5: "What are the player's challenges?" questions the available mechanisms for the developers to create obstacles that separate the player from achieving the gameplay goals. Whether it is simple time restrictions, distractors, or elements with agency, challenges make the game interesting and non-trivial.

Indeed, different language designs can help game developers to define game mental models in different ways. In the case of the PhyDSL-2, its domain-specific language helps game developers to answer the aforementioned questions using five gameplay definition sections: (i) *mobile and static actor definition*, (ii) *environment and layout definition*, (iii) *timed activities definition*, (iv) *scoring rules definition*, and (v) *control definitions*. Let us briefly discuss the how the PhyDSL-2 language has been used to build *AlienMiner*, a physics-based platformer game (Figure 1).

In *AlienMiner* the player controls an alien flying through the world in search of space gems. The player can control a movable avatar using three main control buttons, namely *up*, *right*, and *left*. Scattered throughout the world, there are cannons that shoot fireballs in different directions. The player has to avoid collisions with the fireballs and reach the final

portal where the level is marked as completed. At the end of the game, a final scoreboard is presented summarizing the number of points obtained by the player depending on the number of gems collected.

Listing 1 shows how *AlienMiner's* gameplay model was defined using the PhyDSL-2 language. In summary, games built with PhyDSL-2 are limited to a 2D world ruled by physics, where game actors can be designed to either freely move, or be anchored to coordinates in the world space (Listing 1, 2-8 - see Q1). Furthermore, the player can design the world of the game by determining its gravity, background art, and camera behaviours (Listing 1, 14-17 - see Q3). The games can also include activities dictated by time. Activities can introduce or remove actors to and form the world of the game, and apply physical forces to objects, such as acceleration or friction (Listing 1, 18-21 - see Q5). Also, the games designed with PhyDSL-2 can set rewards using a combination scoring rules that react to three types of events, namely, actor collision events, time events, and on-screen touch events (Listing 1, 22-31 - see Q4). Finally, games can include on-screen controls to manipulate the main actor of the game (if defined) (Listing 1, 32-35 - see Q2). A video of *AlienMiner's* gameplay is available in our research website⁵. Furthermore, in [11] we elaborate on the syntactical support that PhyDSL provides to

⁵AlienMiner Video: <http://goo.gl/WYC8o7>

developers, and its type-system implementation.

B. The PhyDSL-2 Game-template Architecture

Proposition 2: *Game engines impose an architecture to the games that they produce and execute. Consequently, the construction of the game engine is tightly related with the architecture of the game instances that the engine is capable to run. We called this architecture the game-template architecture.* Following Bishop’s [5] and Gregory’s [7] engine-design guidelines and our empirical insights, we defined a set of components that a game engine needs to support, namely, (i) *scene and rendering management*, (ii) *collision and rigid body mechanics*, (iii) *human interface input*, and (iv) *sound and haptic feedback management*. Let us briefly discuss how the components of the PhyDSL-2 *game-template architecture* (Figure 2) support each one of these execution aspects.

The *life-cycle manager* is the main orchestrator of the game mechanics. It controls the main game view at all times (e.g. the score board view, and the gameplay view) and serves as the main runway for the game’s events. The *game-view manager* helps the *life-cycle manager* by storing the state of the game views when they are not visible to the player. For example, the *game-view manager* will maintain a game’s inventory-view up-to-date, while the main action of the game is taking place in the gameplay view.

The *player-input manager* controls the different interaction mechanisms available for the player. These mechanisms range from on-screen controls, to external controlling devices. On the other hand, the *scoring manager* is responsible for the evaluation of the game’s reward and punishment rules. It receives different events generated through the player’s interaction with the game, and determines if the state of the game should change. Changes affect gameplay properties such as player’s lives, number of items available in the inventory, or like in our *AlienMiner* example, the game’s point counter.

The *physics engine* is used to compute the position of all game elements. However, it is the *physics model* the component which understands the meaning of the position of all elements computed by the *physics engine*. The *physics engine* gives physical objects meaning in terms of the different types of gameplay actors; for example, the *physics engine* computes the resulting position of two round objects after a collision, but it is the *physics model* that knows that the two colliding objects are the player’s avatar and a gameplay obstacle.

The *rendering engine* presents a game view to the player, conforming to the *physics model* state. Furthermore, the *sound and haptic feedback controller* manages the available hardware devices to give tangible feedback to the player using, for example, sound and device vibrations.

The *collision detector* filters out the collision events detected by the *physics engine*. It uses the *physics model* to give meaning to the objects involved in a collision, and then notifies the relevant components to change the game state. For example, as a result of a collision event, the *collision detector* notifies the *sound and haptic feedback controller* and

the *scoring manager*. Both components will then determine if they have to emit sounds, or modify the state of the game, to provide feedback to the player.

In the context of PhyDSL-2 template architecture, our main goal was to design a highly cohesive software design; we followed a five-step iterative process to achieve it. First, we implemented concrete components conforming to the architecture described in Figure 2. Then, we identified code segments that together implemented variability mechanisms that could potentially allow developers to create individual titles. We conceptually grouped the variable code segments according to their role in the definition of a gameplay design. This is, each group of code segments implement gameplay variations that solve each of our five gameplay design questions. Finally, we refactored our component implementations, to minimize how scattered each group of code was among the architectural components.

Table I summarizes how the final version of our architecture implements variability mechanisms to answer each one of our five game design questions. While the vertical dimension presents our gameplay design questions, the horizontal dimension describes the different components that comprise the PhyDSL-2 game-template architecture. The intersection of the two dimensions reflects how code segments that implement design variations are related to the architectural components. Furthermore, each row represents a code-segment group that answers to the implementation of a single gameplay-design question. Let us briefly discuss how each row in Table I can be understood as an aspect in aspect-oriented software development, and how we used this separation of concerns strategy to favour the engine’s evolution scenarios.

In aspect-oriented software development [13] [14], an aspect is considered to be a modular representation of a cross-cutting concern. In the game-engine context, a set of code segments that implement a mechanism to answer a game design question can similarly be considered as an aspect. Indeed, taking our gameplay design questions as the game-engine design decomposition strategy, we can identify cross-cutting variability points in our architecture. By grouping them into aspects, we can locate the general portions of code that need to be tailored in order to create variable game implementations, using a fixed architecture [15].

As a concrete example, *S1* and *S2*, are code segments that implement Q1: "Who is the player?" in a gameplay definition. *S1* (located in the *physics model*) allows the engine to specify what is the geometric entity that represents the player’s avatar. *S2* (located in the *rendering engine*) defines the graphical representation of such entity from a graphics point of view, e.g. an image or animation. *S3*, *S4* and *S5* are code segments that together implement Q2: "What are the player’s actions?"; *S3* (located in the *physics model*) allows the engine to define different physical entities, e.g. geometric figures, that represent actionable on-screen controls; *S4* (in the *scoring manager*) enables the engine to specify how the result of control actions translate into different gameplay changes such as point counters, or live counters. Finally, *S5* (located

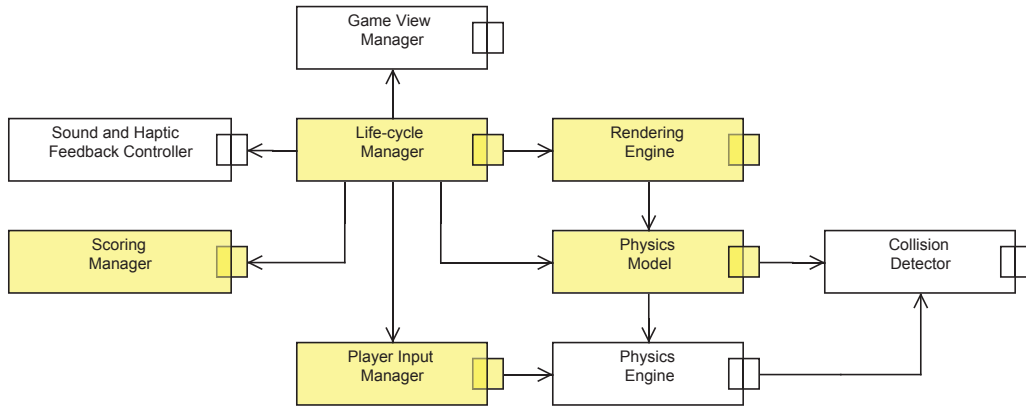


Fig. 2. PhyDSL-2 Game-template Architecture - Variable Components (Highlighted)

TABLE I
PHYDSL-2 GAME PLAY DEFINITION QUESTIONS VS GAME-TEMPLATE ARCHITECTURE COMPONENTS

	Physics Model	Scoring Manager	Rendering Engine	Life-cycle Manager	Player Input Manager
Who is the player?	S1	■	S2	■	■
What are the player's actions?	S3	S4	■	■	S5
Where does the player live?	S6	■	S7	■	S8
What are the player's goals?	S9	S10	S11	■	■
What are the player's challenges?	S12	S13	S14	S15	■

in the *player input manager*) defines what is the effect of the control actions for the player's avatar (if any) e.g. avatar horizontal acceleration, or avatar disappearance.

During the construction of PhyDSL, and its second version PhyDSL-2, we experienced first-hand how the engine scope definition evolves to satisfy the needs of different developers and gameplay mental models. Furthermore, game engines need flexible architectures since they need to react to the requirements and opportunities of new game deployment platforms, e.g. devices with new controllers or sensors, and visual and sound feedback mechanisms [2]. Through the construction process of PhyDSL-2, we became well aware of the fact that, the more scattered the feature implementations are in the architecture, the more likely it is that they get tangled with features to be added. This increases the potential maintenance and evolution costs of the system as a whole [13]. As a concrete example, the introduction of on-screen controls in PhyDSL-2 cross-cuts all the mechanisms of gameplay design specification in PhyDSL. Consequently, we had to redesign how game actors reacted to events, how buttons were included in the game, and how their existence had to be modelled without impacting the behaviour of the other game elements.

One of the main advantages of minimizing how scattered the implementation of gameplay features are is that new gameplay features can be added more efficiently by identifying potential evolution impact points, aligned with the gameplay definition semantics. Consequently, we were able to identify the implementation assets that needed to be maintained, when a new feature needed to be added or fixed. Furthermore, we were able to quickly identify language sections that had to be modified, in order to keep the PhyDSL-2 domain-specific language

with the developer's evolving mental model. In Section II we explore how we have used our separation of concerns strategy, along with model-transformation technologies, to generate gameplay design implementations using a fixed game-template architecture.

II. MODERN MODEL-DRIVEN ENGINEERING

So far we have described how we designed a domain-specific language to capture gameplay mental models for physics-based games. Furthermore, we described how we modularized the architecture of the games supported by the engine, and how we identified aspects that inject variability to the architecture. In this section, we explore how we used model-driven engineering techniques to close the semantic gap between the PhyDSL-2 language, and the game-template architectures, to create individual game instances.

Proposition 3: Model-driven engineering technologies, such as model-to-model and model-to-text transformations, can be used to translate game mental models into executable artifacts, conforming to a reusable game-template architecture. In model-driven engineering, *model-to-model* (M2M) and *model-to-text* (M2T) transformations are used to reduce the level of abstraction of a system specification, until executable artifacts are obtained [16]. Indeed, models captured with domain-specific languages can be used as input of a model transformation to derive executable code.

Using rule-based *model-to-model* transformation technologies, such as ATL [17], the information contained in a model can be split or augmented, thus creating output models whose execution semantics are increasingly specific to the execution infrastructure. Furthermore, using template-based *model-to-text* transformation technologies such as Aceleo [18], models

can be translated into text [16]. Template-based *model-to-text* transformations use meta-code and expansion rules that take information captured in models and generate code segments based on templates prepared for reuse. More often than not, multiple transformations work together in model-transformation compositions to close the gap between a high-level model and the executable artifacts that implement a software specification [19].

In PhyDSL-2, a three step model-transformation composition was used to derive code from a gameplay model expressed using the PhyDSL-2 language (Figure 3). The composition takes the initial specification as input and splits it into four independent game models: (i) *the game layout*, (ii) *the game dynamics*, (iii) *the game scoring*, and (iv) *the game controls*. Similar to object-oriented design principles, in which classes encapsulate behaviours and information in cohesive modules of information, model-transformation compositions can be modularized into self-contained transformation modules [19]. In PhyDSL-2, four independent M2M transformations (i.e. T1, T2, T3, and T4) split the initial gameplay model into four intermediate models that contain information about a specific design sub-domain. This allows five M2T transformation (i.e. T5, T6, T7, T8, and T10) to take the intermediate models as input, and inject a set of aspect implementations into the game-template architecture.

Notice how, in Figure 3, each variable component is derived by one, and only one, intermediate model. This modularization strategy, from both the game-template architecture and the code derivation perspectives, has substantially eased the evolution of the first version of PhyDSL into PhyDSL-2. Particularly, when we introduced the *camera management* and *on-screen control* features into PhyDSL-2, we found that (i) changes were limited to one game-template architecture component, (ii) language changes were progressively made from the impacted intermediate model, to the domain-specific language, thus modifying fewer M2M and M2T transformations, and (iii) developers experienced less severe cognitive challenges while executing maintenance tasks. Particularly, developers of the game engine spent less time trying to understand the entire transformation architecture, and focused their attention on the transformation branch under maintenance.

Proposition 4: Model-transformation compositions are hard to build and maintain. However, modern model-driven development tools enable practitioners to manage, visualize, and maintain transformations, so they can be effectively used in the construction of real software systems. The maintenance and construction tasks around model transformations are expensive, error prone, and cognitively challenging for developers [20]. These challenges stem from the fact that non-trivial model transformations are hard to understand, given that they heavily rely on textual scripts with cryptic operational semantics. Furthermore, as mental models and languages grow in scope and complexity, so do the transformations that derive code from them. Such problems are a barrier for the adoption of model-driven engineering technologies in the general software engineering community.

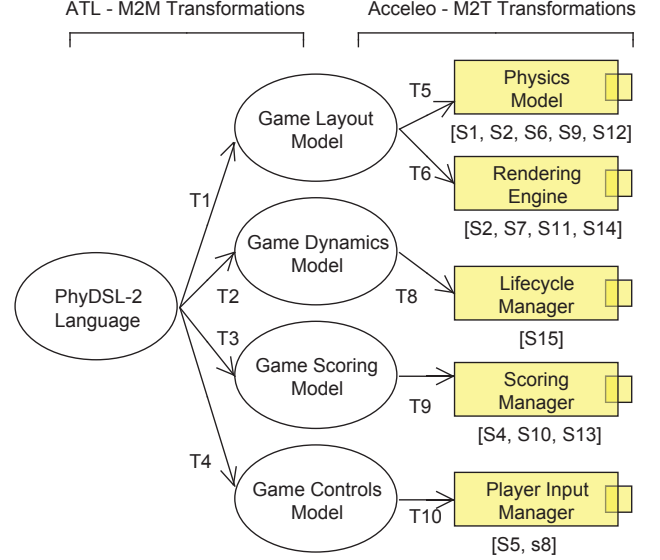


Fig. 3. PhyDSL-2 Game Engine Implementation

To mitigate this problem, we have developed *ChainTracker* [21] [22] a visualization and trace analysis tool that allows developers to visualize models, transformation scripts, and generated architectural components. Figure 4 shows an aggregated view of different visualizations that *ChainTracker* provides for the management of model-driven technologies; among many other features, *ChainTracker* supports the visualization of model-transformation compositions, and provides traceability capabilities to help developers assessing the impact of changes in evolving systems built using MDE technologies. For example, Figure 4 portrays how *ChainTracker* visualizes transformations that generate concrete implementation of PhyDSL-2 *scoring manager* component (i.e T3 and T9 in Figure 3). *ChainTracker* enables developers to keep track of tangled aspects along the architecture implementation, and the transformation composition steps (see Table I and Figure 3). Figure 4 showcases an example of how using annotations, developers can document and automatically isolate transformation modules, model elements, and segments of code, that relate to a specific gameplay design concern. Using *ChainTracker*, we have been able to effectively react to the evolutionary nature of gameplay model definitions, and code implementation requirements. We believe that PhyDSL-2 will soon enable a diverse variety game developers to autonomously define gameplay mental models, and automatically derive executable code in an agile and inexpensive fashion.

III. CONCLUSIONS AND FUTURE WORK

In this paper, we reflected on our experience building PhyDSL-2, a game engine that uses model-driven engineering technologies to capture gameplay models, and derive concrete implementations of 2D physics-based games. The PhyDSL-2 game-specification language was designed to provide a set

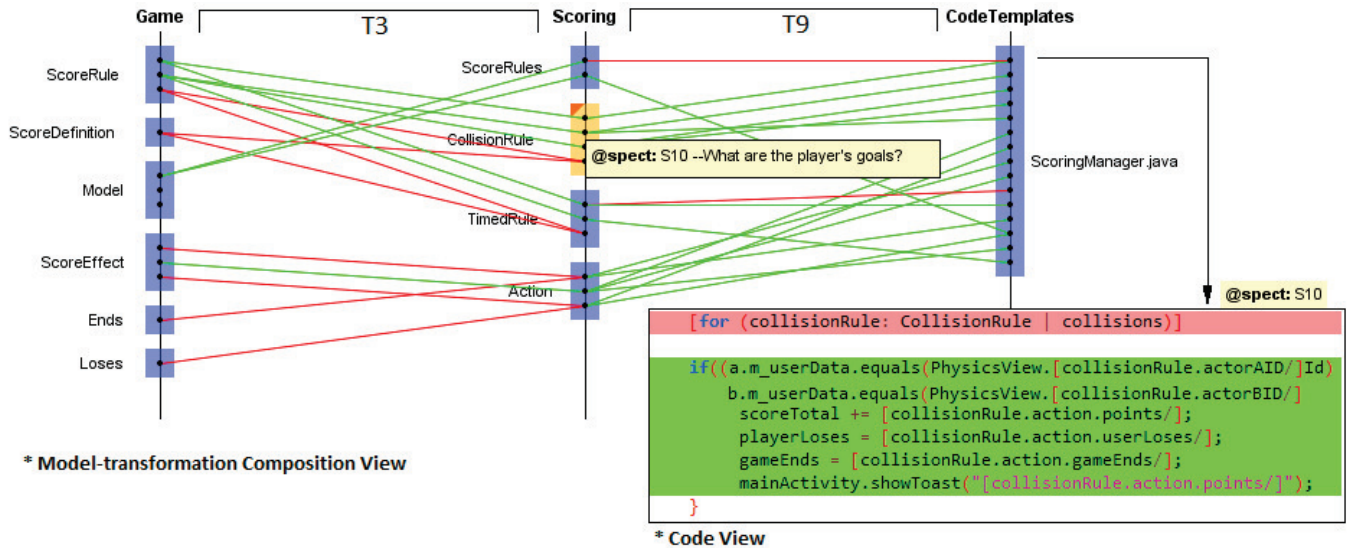


Fig. 4. ChainTracker Environment - Composite View for PhyDSL Game-to-Scoring Branch

of syntactic elements matching a high-level mental model of gameplay with five definition sections. The PhyDSL-2 game engine was developed in a modular manner, so that language changes only affect small scopes of transformation rules. We also briefly discussed how we used *ChainTracker*, in order to manage the complexity and evolution of systems built using model-driven engineering technologies. Through the development and experimentation with PhyDSL-2 and *ChainTracker*, we have empirically validated how model-driven technologies can be effectively used in the construction of game engines. We are now pursuing empirical studies that will increase our understanding of how domain-specific languages benefit game designers in the implementation of gameplay models using high-level semantics, and more accurately evaluate the flexibility of model-driven engineering tools to translate these models into executable artifacts.

REFERENCES

- [1] C. Lewis, J. Whitehead, and N. Wardrip-Fruin, "What went wrong: a taxonomy of video game bugs," in *Proceedings of the fifth international conference on the foundations of digital games*. ACM, 2010, pp. 108–115.
- [2] F. Petrillo, M. Pimenta, F. Trindade, and C. Dietrich, "What went wrong? a survey of problems in game development," *Computers in Entertainment (CIE)*, vol. 7, no. 1, p. 13, 2009.
- [3] C. Lewis and J. Whitehead, "The whats and the whys of games and software engineering," in *Proceedings of the 1st International Workshop on Games and Software Engineering*. ACM, 2011, pp. 1–4.
- [4] J. Blow, "Game development: Harder than you think," *Queue*, vol. 1, no. 10, p. 28, 2004.
- [5] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz, "Designing a pc game engine," *IEEE Computer Graphics and Applications*, vol. 18, no. 1, pp. 46–53, 1998.
- [6] P. N. Johnson-Laird, *Mental models: Towards a cognitive science of language, inference, and consciousness*. Harvard University Press, 1983, no. 6.
- [7] J. Gregory, *Game engine architecture*. CRC Press, 2009.
- [8] M. Văulter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. [Online]. Available: <http://www.dslbook.org>
- [9] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, *End-user development: An emerging paradigm*. Springer, 2006.
- [10] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.
- [11] V. Guana and E. Stroulia, "Phydsl: A code-generation environment for 2d physics-based games," in *2014 IEEE Games, Entertainment, and Media Conference (IEEE GEM)*, 2014.
- [12] K. Salen and E. Zimmerman, *Rules of play: Game design fundamentals*. MIT press, 2004.
- [13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.
- [14] T. Elrad, O. Aldawud, and A. Bader, "Aspect-oriented modeling: Bridging the gap between implementation and design," in *Generative Programming and Component Engineering*. Springer, 2002, pp. 189–201.
- [15] M. Voelter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *Software Product Line Conference, 2007. SPLC 2007. 11th International*. IEEE, 2007, pp. 233–242.
- [16] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. Citeseer, 2003, pp. 1–17.
- [17] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*. Springer, 2006.
- [18] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, "Acceleo user guide," 2006.
- [19] B. Vanhooft, D. Ayed, and Y. Berbers, "A framework for transformation chain development processes," in *Proceedings of the ECMDA Composition of Model Transformations Workshop*, 2006, pp. 3–8.
- [20] V. Guana and E. Stroulia, "Backward propagation of code refinements on transformational code generation environments," in *Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop on*, 2013, pp. 55–60.
- [21] V. Guana, K. Gaboriau, and E. Stroulia, "Chaintracker: Towards a comprehensive tool for building code-generation environments," in *Proceedings of the 2014 International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Press, 2014.
- [22] V. Guana and E. Stroulia, "Chaintracker, a model-transformation trace analysis tool for code-generation environments," in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8568, pp. 146–153.